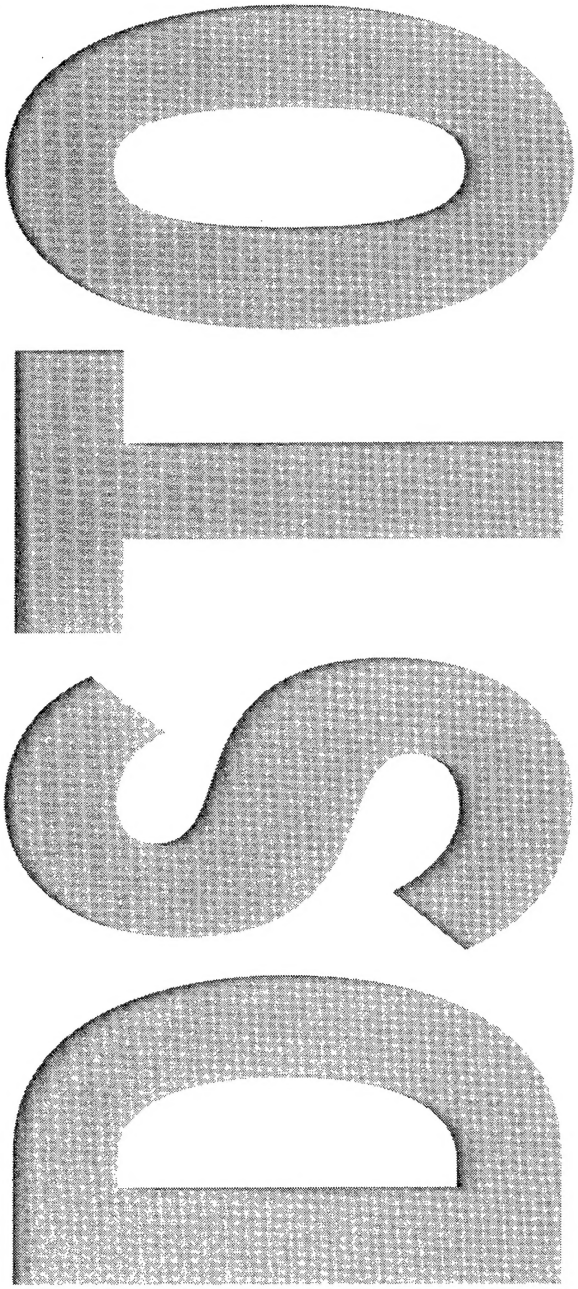
Apr 2003



Evaluation of Methods for Rapidly Inserting Data into an Oracle Relational Database

Mark I. Porter and Ian L. Coat

DSTO-TN-0487

Evaluation of Methods for Rapidly Inserting Data into an Oracle Relational Database

Mark I. Porter and Ian L. Coat

Intelligence, Surveillance and Reconnaissance Division
Information Sciences Laboratory

DSTO-TN-0487

ABSTRACT

In the Intelligence, Surveillance and Reconnaissance Division (ISRD) there are numerous applications in which large volumes of data are collected and analysed. Often, relational databases are used to store such data, as they improve certain analytical tasks through querying. In the literature, there is ample documentation of methods by which queries on a database may be optimised. However, there is scant information on the optimal technique for the initial insertion of data at high speed. This paper addresses the problem by investigating numerous insertion methods and comparing their performance for a given data set. The work was carried out on standard PCs running a commercially available database product (Oracle) and using common languages (such as Java, C and PL/SQL), and therefore may be of interest to the wider DSTO community.

Published by

DSTO Information Sciences Laboratory
PO Box 1500
Edinburgh South Australia 5111 Australia

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR- 012-722
April 2003

APPROVED FOR PUBLIC RELEASE

Evaluation of Methods for Rapidly Inserting Data into an Oracle Relational Database

Executive Summary

Relational databases are a useful means of storing data in a convenient form for querying and selecting specific subsets from that data. In the Intelligence, Surveillance and Reconnaissance Division (ISRD) there are numerous applications that generate large volumes of data that need to be collected and analysed, and the best way to store them is in such databases.

The efficient querying of databases, once the content has been inserted, is a well-researched area. Guidance in the literature on the optimal method by which to initially insert the data is less common, and often is merely a rule of thumb. It is the aim of this technical note to evaluate the performance of different methods of inserting data into a relational database.

Defence has a licensing agreement with Oracle by which their relational database product may be used at no cost. Consequently, this is the database product used in this investigation. The insertion methods employed are therefore those that are compatible with such a database. These include Java, Pro*C, OCI, PL/SQL and others. The applications for which this work is tailored require easily deployable systems, and therefore desktop PCs have been used. It is thought that the use of a common database product, coupled with widely used languages and instituted on PCs is of general applicability, and therefore may be of interest to the wider DSTO community.

It was found that the optimal method for rapid insertion used bulk binding in prepared statements with infrequent COMMIT commands, using either Pro*C or OCI.

Contents

1. Introduction

In the Intelligence, Surveillance and Reconnaissance Division (ISRD) of DSTO, there are many applications which generate large amounts of data at high speed in a continuous stream. Such data can be most useful if it is stored in a database and later queried. The results of these queries may be used to modify the data collection process. Under these circumstances, the insert and query processes need to be done in real time.

A survey of the literature shows there to be ample guidance on optimisation of query retrieval from a database (for example, Kamel and King (1992), Goetz (1993), O'Neil and Quass (1997), Armstrong (2001)), but minimal advice is available on the initial insertion of data. The aim of this paper is to identify various rapid insertion techniques and evaluate them to find the optimal method.

The practicalities of the situation for ISRD dictate that the system be deployed on easily moveable equipment, and therefore commercial PCs are being used. Due to its scalability and a current licensing agreement, the commercial database software employed is Oracle. This use of common computer hardware and software that is freely available to Defence results in this research being applicable to a wider DSTO audience.

The focus of this paper is on the methods which deliver the fastest results, rather than the optimal configuration of the Oracle database. Therefore, minimal attention has been paid to this aspect, as it is assumed that any performance improvements made within the database would be cumulative to the efficiency of the optimal insertion method. The database parameters were mostly unchanged from the "default" settings at creation time.

The major goal of this technical note is to establish an optimal method for inserting data into a database. In practice, the inserted data will later be queried. Methods which optimise data insertion may be detrimental to querying, and vice versa. Adding an index to a table, for example, greatly increases the query efficiency, but also increases the time required to add data to the table. Such coupling effects lie outside the scope of the current research.

2. Experimental Set up

2.1 Hardware

The Oracle database resided on a Windows 2000 server with a 2.2GHz Pentium 4 processor, having 1GB RAM. In some cases a client machine was needed, in which case a similar machine was used, but running Windows XP as the operating system. A 100 Mbps link between the two machines was used in this instance.

2.2 Software

The Oracle database used was version 8.1.6. The Oracle utilities used, such as Pro*C, OCI, SQL*Loader and SQLJ, were all distributed with this version of Oracle.

In cases where Java was needed, Java 2 Version 1.4 was used. Oracle's OCI8 JDBC driver was used in these instances. This is a type 2 JDBC driver.

The C compiler used in the investigation was LCC (http://www.cs.virginia.edu/~lcc-win32), which is freely available.

2.3 Date Set

The data inserted into the database was purely artificial, but simulated a non-normalised dataset. In the case considered here, the normalised dataset *would* have consisted of a primary table of 100 fields with a primary key and a foreign key that corresponded to the primary key of a secondary table with 10 fields. In the non-normalised case, however, the sole table has 109 fields (as the foreign key is not included twice).

The dataset contains 10,000 records, which are all numeric in type. Prior work by the authors has indicated that the insertion rate is virtually independent of the dataset size. Hence, the data size chosen for this work is large enough for insert operations to dominate over other background database management system (DBMS) processes, but not so large that insertion times become prohibitive.

The first field in the table is the primary key (although this is not defined to the database, to reduce background processes) and ranges through the integers from 1 to 10,000. The second field, which would correspond to the foreign key had the table been normalized, cycles from 1 through to 10. If the records are arranged in ascending order of the primary key, then the value of the n^{th} record in the i^{th} column, R_{in}, is given as:

$$R_{in} = 100{,}000 + 100i + n \qquad \text{for } 3 \le i \le 100 \qquad \text{(Equ. 1)}$$

and

$$R_{in} = 200{,}000 + 100(i - 99) + \text{rem}(n \div 10) \qquad \text{for } 101 \le i \le 109 \qquad \text{(Equ. 2)}$$

where $\text{rem}(n \div 10)$ is the remainder of n divided by 10.

The resulting dataset is 6,318K of "raw" data, which rises to 8,477K when it is written to a well-formatted text file (see note for file type S in Additional Notes on Table 1, Note 3). The data was inserted "as is", without the introduction of a primary key or indices, and the final form of the data in the database was independent of the method by which it was inserted.

2.4 Performance metrics

It is often the case that the number of input/output (I/O) operations involved in a transaction is used as the performance metric for SQL queries (for example, Rahayu *et al.* (2001)). This measurement criterion is not applicable to the current study, as the number of disk writes is constant, due to the common dataset being inserted. Therefore, the time to insert the dataset is taken as the performance metric. Note that the throughput may also be used, but this value is a derivative of the execution time, and therefore not a primary, measured, indicator.

2.5 Insertion techniques

This investigation is aimed at providing results that are of relevance to the problem of continuous data insertion into the database. Two main methods were identified by which the transmission of data between data-generating application and database could be achieved. The first method involves the source program writing the data to file periodically. The file would then be accessed by the database at regular intervals. Alternatively, the source program could communicate directly with the database, sending SQL INSERT statements as the data is generated. These two broad methods are reflected in the types of insertion techniques used in the investigation. Note that, in the file writing and reading case, there are numerous hardware related modifications which could improve performance, such as the use of a RAM disk for storing the temporary file. Such measures are outside the scope of this study.

A number of languages and utilities were used in the survey of insertion methods available in conjunction with an Oracle database. They include PL/SQL, Pro*C, OCI, SQL*Loader, Java and SQLJ. A summary of each is given in Appendix A. In addition, Appendix B gives sample code for Pro*C and OCI.

Despite the use of numerous languages, there are certain techniques which are widely supported in generating and executing SQL statements. These are detailed below. Table 1 lists the tests performed, and the methods used in each.

Table 1: Methods of insertion

Test number	Language/ utility	Insertions per COMMIT	Local (L) or Client (C)	Concatenation	Binding	Bulk binding	Prepared statement	File (F) or generated (G)	File type	APPEND hint	Connection speed (Mbps)	Batch size
1	PL/SQL	1	L	X				G				
2	PL/SQL	500	L	X				G				
3	PL/SQL	1000	L	X				G				
4	PL/SQL	10,000	L		X			F	C			
5	PL/SQL	10,000	L		X	X		F	C			
6	PL/SQL	1	L		X			G				
7	PL/SQL	1	L		X			G		X		
8	PL/SQL	10,000	L		X			G				
9	PL/SQL	10,000	L		X			G		X		
10	PL/SQL	10,000	L		X	X		G				
11	PL/SQL	1	L		X			F	S			
12	PL/SQL	10,000	L		X			F	S			
13	PL/SQL	10,000	L		X			F	S	X		
14	PL/SQL	10,000	L		X	X		F	S			
15	PL/SQL	10,000	L		X	X		F	S	X		
16	PL/SQL	10,000	L		X	X		F	V			
17	PL/SQL	10,000	L		X	X		F	S			
18	SQL*Loader	10,000	L					F	C			
19	SQL*Loader	10,000	L					F	C			
20	SQL*Loader	10,000	L					F	S			
21	SQL*Loader	10,000	L					F	S			
22	Pro*C	10,000	C		X			G			10	
23	Pro*C	10,000	C		X			G			100	
24	Pro*C	10,000	C		X	X		G			100	
25	Pro*C	10,000	C		X	X	X	G			100	
26	JDBC	Auto	C	X				G			100	
27	JDBC	Auto	C		X		X	G			100	
28	JDBC	10,000	C		X		X	G			100	
29	JDBC	10,000	C		X		X	G			100	10
30	JDBC	10,000	C		X		X	G			100	10
31	JDBC	10,000	L		X		X	G				10
32	SQLJ	10,000	C		X			G			100	
33	SQLJ	Auto	C		X		X	G			100	
34	SQLJ	10,000	C		X		X	G			100	
35	SQLJ	10,000	C		X		X	G			100	10
36	SQLJ	10,000	C		X		X	G			100	
37	OCI	10,000	C		X		X	G			100	
38	OCI	10,000	C		X	X	X	G			100	

Additional notes on Table 1

1. *Local or Client* indicates whether the source data application was executed on the same machine as the database (local) or over a network (client).
2. *File or Generated* refers to the data source. In the case of File, the data was read from a file and inserted into the database. Generated means that the application generated the data during execution.
3. The three file types used were: C – comma separated values, S – space separated values with the data arranged in columns, and V – vertical file with one value per line.
4. The *Connection speed* given was the capacity of the network connection between client and server machine.
5. The entry *Auto* in the *Insertions per COMMIT* field signifies that "Autocommit" was in operation, which caused automatic commits after each insert statement.
6. For SQLJ tests, *Prepared statement* means statement caching was used.
7. In Test 19 the format of the data being read was specified as explicitly as possible.
8. In Test 20 WHITESPACE was used as the data delimiter.
9. In Test 21 the data columns were defined by their position, rather than using a field separator.
10. In Test 22 the sizes of the buffer and string used to read the data were defined as the minimum possible, as were the NUMBER variables into which the data was passed.
11. In Test 29 Java batching was used.
12. In Tests 30 and 31 Oracle's Java batching was used.
13. In Test 36 SQLJ parameter size hints were used, which provide hints to the DBMS about the parameters being passed.

2.5.1 String concatenation

The following code segment demonstrates string concatenation used to generate dynamic SQL, where `a1` to `an` are variables which are modified to provide varying statements.

```
"INSERT INTO table (field1,..,fieldn) VALUES (" + a1 + .. + an +
")"
```

2.5.2 Bind variables

In place of re-forming the string each time as in the concatenation method above, the use of bind variables allows the process to be automated. In the code extract below, the variables `a1` through `an` are bound to the string, and evaluated by the host program at runtime.

```
"INSERT INTO table (field1,..,fieldn) VALUES (:a1,..,:an)"
```

2.5.3 Bulk binding

Also known as bulk processing, this technique allows arrays of data, rather than single values, to be passed to the database. The SQL statement is then evaluated for every value in the arrays. The benefit of such a method is the reduction in calls to the SQL engine, as a mass of data is passed with one SQL call.

```
"INSERT INTO table (field1,..,fieldn) VALUES
(:array1,..,:arrayn)"
```

2.5.4 Prepared statements

In cases where a SQL statement is executed repeatedly, it can be advantageous to "prepare" the statement in advance. In contrast to the normal SQL statement, which is compiled numerous times, the prepared statement is only compiled once. It is then the compiled statement which is re-run, thus reducing the amount of processing. The code extract below shows the general form of a prepared statement, and its execution. In this case, the variables `a1` to `an` are merely placeholders. When the `EXECUTE` command is invoked, the values `array1` to `arrayn` are substituted in these places. Note that in this case these latter variables are arrays, leading to bulk binding being used in the prepared statement context.

```
Prepare("INSERT INTO table (field1,..,fieldn) VALUES
(:a1,..,:an)");
EXECUTE prepared_statement USING :array1,..,:arrayn;
```

2.5.5 Batch processing

In bulk binding, a mass of data is sent to the SQL engine with an associated SQL statement. In batch processing, a series of statements are buffered and periodically sent to the engine as batches. This reduces overhead associated with calling the SQL engine.

2.5.6 Appending data

When new data is being inserted into the database, a suitable location must be found to store it. To do this the currently used data blocks on the disk are scanned for free space. If there is existing capacity, then the data is added within the existing blocks. In the event that there is no available space, the new data is appended in the new space allocation. It is possible to specify that all data should be appended in this manner, to eliminate the overheard associated with searching for space. In SQL*Loader, this directive is specified at the command prompt, while for SQL based methods it can be done within the SQL statement itself using a hint. Hints take the form of comments within the SQL string which are interpreted by the DBMS.

3. Results

The results in the following figures are indexed by the test number given in Table 1. The total length of the bars in the charts represents the average time taken to insert the 10,000 record dataset discussed in Section 2.3. A minimum of 5 repetitions was performed for each test. In some cases, this measurement is subdivided into portions labeled *Database* and *CPU*. These are nominal descriptions, as the *CPU* measure is the amount of time taken for the application to execute without actually performing the insert operation (the INSERT statements being commented out in these cases). That is, it is the time to read or generate, and then pre-process the data ready for insertion. The *Database* portion is the

total time less the *CPU* time, thus indicating the time taken to execute the insert operations, which is indicative of the performance of the database.

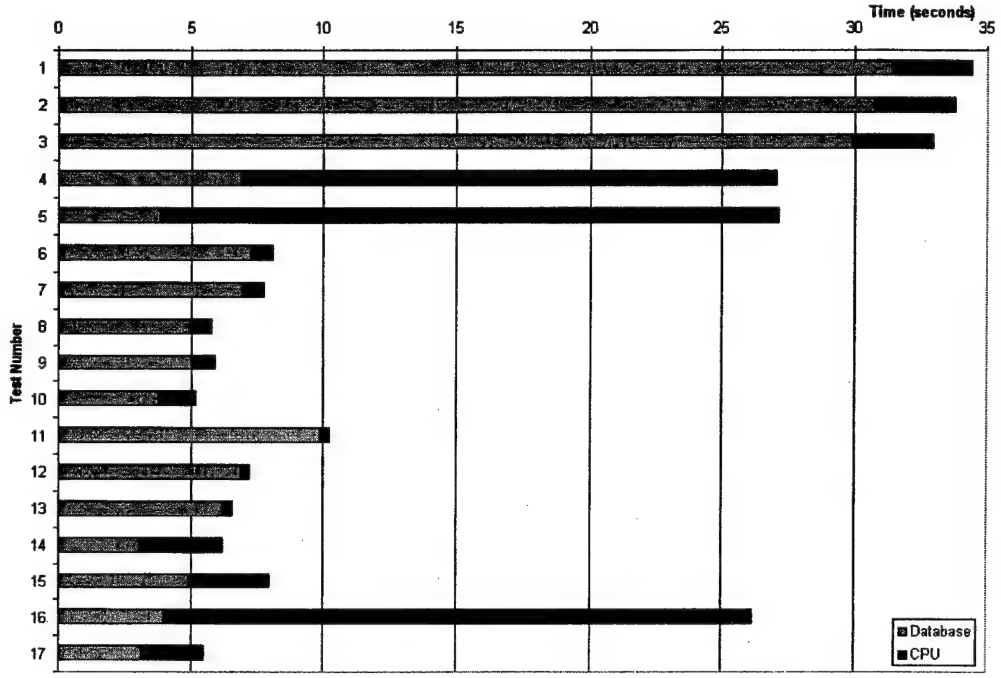


Figure 1: Insertion using PL/SQL

Figure 1 gives the results for all methods that used PL/SQL. It is immediately obvious from the graph that string concatenation (Tests 1-3) performs poorly. The three bars also show that decreasing the frequency of the COMMIT command reduces execution time, but this improvement is slight compared to the dominance of the concatenation inefficiency. Due to this obvious penalty, string concatenation was rarely used in subsequent tests.

Tests 4 and 5 in Figure 1 illustrate that the efficiency of the database at inserting data is not necessarily the bottleneck in all cases. In these tests, total execution time is dominated by the *CPU* or processing time. A comma separated file was used as the data source for these tests, and the poor processing performance of this data is due to the lack of efficient PL/SQL routines to deal with data formatted in such a way. Tests 11 to 17 show that a marked performance improvement may be achieved through the use of positional data files (file type S) in which delimiters are not used to demarcate values. The input data file for Test 16 was "vertical", signifying that each value was on a new line. This eliminated the need for pre-processing of each line, but increased the number of lines read. The result presented here clearly indicated that it is the line read that is the less efficient of these two operations.

Tests 6 to 10 did not have an input data file associated with them. Instead, the data to be inserted was generated within the application, which also formed the SQL statements and called the database. Comparing Test 6 with 7 and Test 8 with 9 reveals that the APPEND

hint in the SQL string had little effect on the performance. Indeed, from these results it is not possible to determine whether or not this method is generally beneficial. Test 10 introduces bulk binding and shows it to be the most efficient method with PL/SQL.

Figure 2 displays the results for the SQL*Loader utility, OCI and Pro*C. All three methods are provided by Oracle, although the latter two are written and executed in an external C environment.

SQL*Loader was used to load a comma separated file in Tests 18 and 19. The difference between the two bars shows that the performance improves with strict definition of the data being loaded. However, these two tests took longer to execute than Tests 20 and 21, which loaded files formatted without commas. Of the SQL*Loader results, the optimal performance was achieved in Test 21 where the data was specified by its position on the line, rather than through the use of a delimiter. It is important to note that the *Database* and *CPU* segments of the bars for the SQL*Loader tests are not directly comparable with those of other tests. In the SQL*Loader case, the elapsed and CPU time were outputs of the utility. It is not clear whether the CPU time given in this case is equivalent to the CPU time measured for the other insertion methods.

Client/server architecture was first used in Test 22. The transmission of data between machines introduced a new impediment to performance. Indeed, it was found that increasing the connection speed from 10 to 100 Mbps markedly reduced the execution time (see Tests 22 and 23). The major efficiency improvement achieved with Pro*C was the use of bulk binding in Test 24. Figure 2 shows that a further, if less substantial, improvement was gained with a prepared statement (Test 25).

Both Pro*C and OCI are developed in a C environment. In Figure 2, Tests 37 and 38 show the performance of OCI in comparison with Pro*C. Based on the methods used, Test 37 is the OCI equivalent of Test 23, while Test 38 is most like Test 24. The trend of decreasing execution time with bulk binding is again observed with OCI. The most efficient OCI method (Test 38) uses the same method as the optimal Pro*C method (Test 25), and achieves approximately the same execution time.

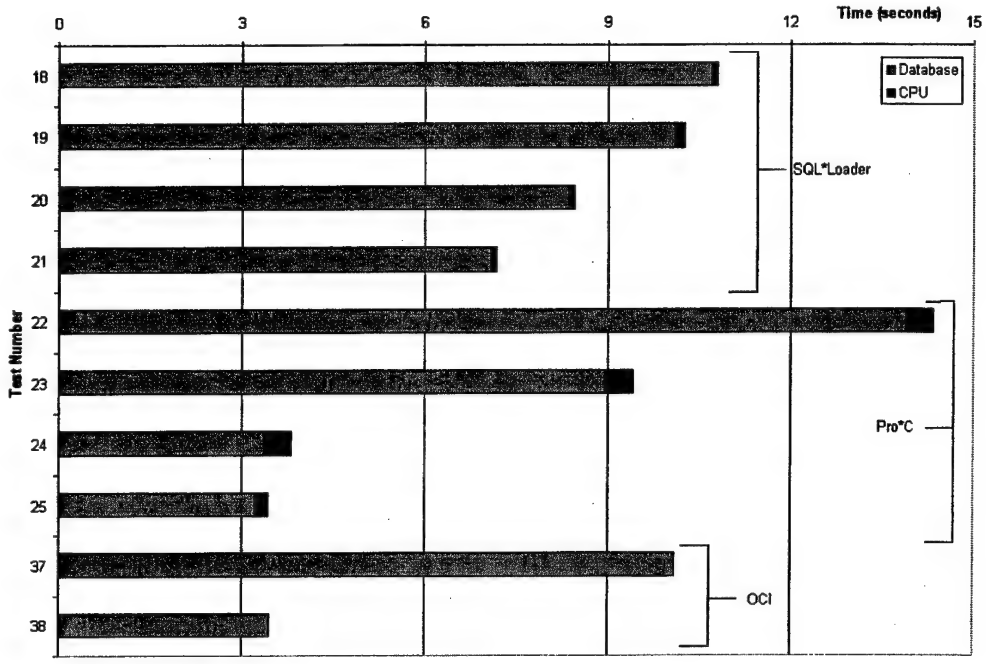


*Figure 2: Results for SQL*Loader, OCI and Pro*C*

Figure 3 presents the results for all the tests based on Java. These are Tests 26 to 36, which encompass the results using JDBC and SQLJ. Test 26 demonstrates the result found earlier that using dynamically created SQL strings with concatenation is an inefficient method. It was also shown earlier that frequent COMMIT operations detract from performance, as reiterated by comparing Test 27 with Test 28. Test 29 introduces batching, discussed in Section 2.5.5. In this case, it is the standard batching function given in the JDBC API. Figure 3 shows that JDBC batching is detrimental to performance, which is somewhat surprising, as it is designed as a performance improvement. The authors surmise that the execution time of Test 28 is near to the optimal time achievable using Java. If this is the case, then the addition of batching in Test 29 only adds pre-processing expense, while being unable to further improve performance. Tests 30 and 31 show that neither the use of Oracle's own batching procedure nor the running of the program on the local server machine can improve on the performance of Test 28. It is also worth noting that this time, of approximately 16.5 seconds, is a fourfold increase on the execution time possible using Pro*C. This may be partly due to the higher level of interpretation involved in using Java, but may also indicate that another bottleneck is at work. The default character set used by Oracle is a superset of ASCII, not Unicode on which Java is based. A new database was established with a Unicode native character set, and the Java tests repeated. The results were unchanged, indicating that the character set conversion is not the rate-determining step for Java.

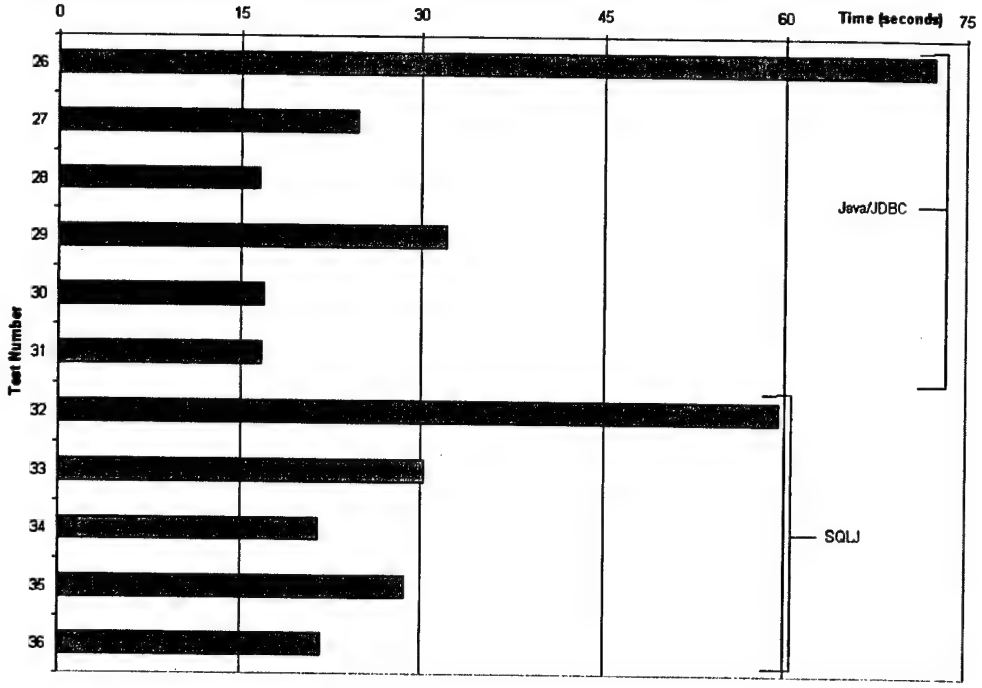


Figure 3: JDBC and SQLJ results

Tests 32 to 36 used SQLJ instead of JDBC calls to interact with the database. Figure 3 again shows that batching has no beneficial effect on performance. It is worth noting that there is no simple way of processing arrays for bulk binding in either JDBC or SQLJ. It is possible to pass an array using JDBC, but it is a convoluted process, and therefore inefficient. Figures 1 and 2 show bulk binding to be highly advantageous, and explains the superiority of Pro*C, OCI and PL/SQL over Java based applications when optimising for insertion speed.

4. Conclusion

It is often assumed that the major bottleneck to database performance is I/O operations (Jalics and McIntyre (1989), Hsu *et al.* (2001)). Indeed, it is true that eliminating these, for example by using the cache in querying operations, does lead to lower execution times. However, the current work has shown that even in intensive I/O transactions, where all data is written to disk, optimisation of the application method can have a significant impact on the achievable throughput. In the case of rapid insertion considered here, it was found that the choice of language and insertion methods could decrease execution time by a factor of 10.

The optimal method, of those surveyed, used bulk binding with a prepared statement in Pro*C or OCI. Sample code for each of these cases has been supplied in Appendix B. In addition, the following conclusions can be drawn. Measures such as binding, bulk processing and preparing statements have significant impacts as they reduce the number of calls to the SQL engine during execution. Other measures, such as using well-formed input files and accurately specifying parameter sizes, reduce pre-processing and therefore also improve performance.

The results presented here are transferable to other applications where high speed insertion is of importance. These might include telephone logging, competition entries, or any other event in which minimal subsequent updating of the data will occur.

5. References

ARMSTRONG, R. (2001), Seven steps to optimising data warehouse performance, *Computer*, **34**(12):76-79.

GOETZ, G. (1993), Query evaluation techniques for large databases, *ACM Computing Surveys (CSUR)*, **25**(2):73-169.

HSU, W.W., SMITH, A.J. AND YOUNG, H.C. (2001), I/O reference behaviour of production database workloads and the TPC benchmarks – An analysis at the logical level. *ACM Transactions on Database Systems*, **26**(1):96-143.

JALICS, P.J., MCINTYRE, D.R. (1989), Caching and other disk access avoidance techniques on personal computers, *Communications of the ACM*, **32**(2):246-255.

KAMEL, N., KING, R. (1992), Intelligent database caching through the use of page-answers and page-traces, *ACM Transactions on Database Systems*, **17**(4): 601-646.

O'NEIL, P., QUASS, D. (1997), Improved query performance with variant indexes, *International Conference on Management of Data and Symposium on Principles of Database Systems*, Proceedings of the 1997 ACM SIGMOD international conference on Management of data, 38-49.

ORACLE CORPORATION (2000a), Oracle Documentation Library, Oracle 8i Server Application Development, Release 8.1.6, Oracle8i Application Developer's Guide, Release 2 (8.1.6), Ch. 1 Understanding the Oracle Programmatic Environments, A76939-01.

ORACLE CORPORATION (2000b), Oracle Documentation Library, Oracle 8i Server and Data Warehousing, Release 8.1.6, Oracle8i Utilities, Release 2 (8.1.6), Ch. 8 SQL*Loader: Conventional and Direct Path Loads, A76955-01.

RAHAYAU, J.W., CHANG, E., DILLON, T.S. AND TANIAR, D. (2001), Performance evaluation of the object-relational transformation methodology. *Data & Knowledge Engineering*, **38**:265-300.

Appendix A: Description of Languages and Utilities Used

A.1. PL/SQL

This is Oracle's built in scripting language. Scripts can be written and executed from within an Oracle environment. PL/SQL benefits from its integration with the database, which often results in faster execution times (Oracle Corporation, 2000(a)).

A.2. SQL*Loader

SQL*Loader is an Oracle utility which enables flat data files to be read directly into the database, thus circumventing much of the processing which may otherwise normally occur. According to Oracle's documentation (Oracle Corporation, 2000(b)), the writing of the data to disk takes place at near disk speed. All results presented here used the direct load option which, in a similar way to the Append hint of Section 2.5.6, appends data directly to new disk area, rather than searching for free space in existing data blocks.

A.3. OCI

With the Oracle Call Interface (OCI) application programming interface (API), C programs can interact with an Oracle server and perform operations on the database. The resulting C code does not need to be pre-compiled, as for Pro*C and SQLJ. Instead, OCI functions are invoked in the code, and accessed through a link to the OCI library.

A.4. Pro*C

Using Pro*C, SQL statements can be embedded within C code. The hybrid code is precompiled with Pro*C to generate a C file, which is then compiled and run as normal.

A.5. JDBC

Through the JDBC API, Java is able to communicate with relational databases. It is Java equivalent of OCI, where no precompilation is necessary, but library functions are used instead of embedded SQL statements.

A.6. SQLJ

SQLJ enables Java in the same way that Pro*C does for C, embedding SQL statements within the parent program. The SQLJ precompiler then generates Java code, which is compiled and executed as normal.

Appendix B: Example Computer Code for Selected Insertion Techniques

B.1. Sample Pro*C code

The following abridged code extract (proc_demo.pc) is a Pro*C programme which drops and then creates the table for the data, generates the data and inserts it using a prepared statement with bulk binding. The programme must first be precompiled using the Oracle Pro*C Precompiler (to proc_demo.c for example) before being compiled and run as a normal C programme. The lines beginning with EXEC contain the embedded SQL statements. Note that the data is generated within the programme, and therefore is included in the execution time measured. This section of the code executes significantly faster than the database portion, and therefore does not impact greatly on the measurements.

proc_demo.pc

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlca.h>
#include <string.h>
#include <time.h>

void sql_error(){     //Error handling
      char msg[200];
      size_t buf_len, msg_len;

      buf_len=sizeof(msg);
      sqlglm(msg,&buf_len,&msg_len);
      printf("%.*s\n\n",msg_len,msg);
      exit(1);
}

void main(void){
char username[]="usrnm";
char password[]="pswrd";
char conn[] = "db01.dsto.defence.gov.au";

int N=10000, Narray=2000, N2=10, a[109][Narray], k;
float c_init,c_time;

EXEC SQL BEGIN DECLARE SECTION;
char insert_stmt[1500];

VARCHAR usr[50];
VARCHAR passwd[50];
varchar db_string[50];
int i,i2,j,count;

int c1[2000],c2[2000],c3[2000],..,c108[2000],c109[2000];
```

```
    EXEC SQL END DECLARE SECTION;

    strncpy(usr.arr, username, strlen(username));
    usr.len = strlen(username);

    strncpy(db_string.arr, conn, strlen(conn));
    db_string.len = strlen(conn);

    strncpy(passwd.arr, password, strlen(password));
    passwd.len = strlen(password);

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    /* Connect to the database */
    EXEC SQL CONNECT :usr IDENTIFIED BY :passwd USING :db_string;

    EXEC SQL DROP TABLE PENTOPNON;  //Drop and create table
    EXEC SQL CREATE TABLE PENTOPNON (pk1 NUMBER(9),
    pk2 NUMBER(5),field3 NUMBER(10),..,field99 NUMBER(10),
    field100 NUMBER(10),bfield2 NUMBER(10),..,
    bfield9 NUMBER(10),bfield10 NUMBER(10));

    /* Generate INSERT statement */
    strcpy(insert_stmt,"INSERT INTO PENTOPNON (pk1,pk2,field3,
    field4,..,field99,field100,bfield2,..,bfield10)
    VALUES (:c1,:c2,..,:c108,:c109)");

    EXEC SQL PREPARE sql_stmt FROM :insert_stmt; //Prepare statement

    c_init=clock()/1000.0;  // Record time at start

    /* Generate the data assign it to the 2D array a[][] */
    j=1;
    for(i2=1;i2<=N;i2++){
          i=((i2-1)%Narray)+1;
          a[0][i-1]=i2;
          a[1][i-1]=j;
          for (k=2;k<=99;k++){
               a[k][i-1]=100000+100*(k+1)+i2;
               if(k<=10){
                    a[k+98][i-1]=200000+100*k+j;
                    }
          }
          /* Now assign the data to individual vectors associated
    with each variable in the prepared statement */
          c1[i-1]=a[0][i-1];
          c2[i-1]=a[1][i-1];
          ..
          ..
          c108[i-1]=a[107][i-1];
          c109[i-1]=a[108][i-1];

          j++;
          if(j>N2){
               j=1;  //Reset counter for the repeating data values
               }

    /* Execute the prepared statement with the data from the vectors
    above when Narray data records have been generated */
    if(i==Narray){
    EXEC SQL EXECUTE sql_stmt USING c1,:c2,..,:c108,:c109;
```

```
}
}
EXEC SQL COMMIT;  // Commit the insertions

/* Record the time and print the elapsed time */
c_time=clock()/1000.0;
printf("%10.3f\n",c_time-c_init);
}
```

B.2. Sample OCI code

The following OCI program is written in C and does not require precompilation as for Pro*C. The code has been condensed for brevity.

oci_demo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <oci.h>  //OCI libraries
#include <OCIAPR.H>
#include <OCIDFN.H>
#include <ORATYPES.H>

// OCI environment and handle variables
static OCIEnv           *p_env;
static OCIError         *p_err;
static OCISvcCtx        *p_svc;
static OCIStmt          *p_sql;
static OCIDefine        *p_dfn    = (OCIDefine *) 0;
static OCIBind          *p_bnd    = (OCIBind *) 0;

static int N=10000; // Number of records to be inserted
static int N2=10;
static int Narray=200; // Size of buffer array
static int Ncommit=1000; // Number of records per COMMIT

void main(){
int  p_bvi,rc,p_int,errcode,i,i2,j,k,count,a[109][200];
char p_sli[20],errbuf[100],sql_string[197];
float c_init,c_time;
FILE *fp;

fp = fopen("testfile.txt","w");

/* Initialize OCI environment and handles */
rc = OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
     (dvoid * (*)(dvoid *, size_t)) 0,
     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
     (void (*)(dvoid *, dvoid *)) 0 );
rc = OCIEnvInit( (OCIEnv **) &p_env, OCI_DEFAULT, (size_t) 0,
     (dvoid **) 0 );
rc = OCIHandleAlloc( (dvoid *) p_env, (dvoid **) &p_err,
     OCI_HTYPE_ERROR,(size_t) 0, (dvoid **) 0);
```

```
    rc = OCIHandleAlloc( (dvoid *) p_env, (dvoid **) &p_svc,
         OCI_HTYPE_SVCCTX,(size_t) 0, (dvoid **) 0);
    rc = OCIHandleAlloc( (dvoid *) p_env, (dvoid **) &p_sql,
         OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0);

    /* Connect to database server */
    rc = OCILogon(p_env, p_err, &p_svc, "usrnm", 5, "pswrd", 5,
         "db01.dsto.defence.gov.au", 24);
    if (rc != 0) {
         OCIErrorGet((dvoid *)p_err, (ub4) 1, (text *) NULL,
         &errcode, errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
         printf("Error - %.*s\n", 512, errbuf);
         exit(8);
    }

    /* Drop the table */
    strncpy(sql_string, "drop table pentopnon", 20);
    /* Prepare drop table statement */
    rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 20,
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    /* Execute the SQL statment */
    rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
         (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
         OCI_DEFAULT);

    /* Drop the index */
    strncpy(sql_string, "drop index pk1_pentopnon", 24);
    /* Prepare drop table statement */
    rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 24,
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    /* Execute the SQL statment */
    rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
         (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
         OCI_DEFAULT);

    /* Create the table string */
    strncpy(sql_string, "CREATE TABLE PENTOPNON (pk1 NUMBER(10),pk2
    NUMBER(5),field3 NUMBER(10),field4 NUMBER(10),field5
    NUMBER(10),field6 NUMBER(10),field7 NUMBER(10),field8
    NUMBER(10),field9 NUMBER(10),field10 NUMBER(10)) TABLESPACE
    users PCTFREE 10 PCTUSED 40 STORAGE ( INITIAL 128K NEXT 10240K
    PCTINCREASE 0)", 287);
    /* Prepare create table statement */
    rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 287,
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    /* Execute the SQL statment */
    rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
         (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
         OCI_DEFAULT);

    strncpy(sql_string, "ALTER TABLE PENTOPNON ADD (field11
    NUMBER(10),field12 NUMBER(10),field13 NUMBER(10),field14
    NUMBER(10),field15 NUMBER(10),field16 NUMBER(10),field17
    NUMBER(10),field18 NUMBER(10),field19 NUMBER(10),field20
    NUMBER(10))",217);
    /* Prepare create table statement */
    rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 217,
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    /* Execute the SQL statment */
    rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
         (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
```

```
      OCI_DEFAULT);
      ..
      ..
      ..
strncpy(sql_string, "ALTER TABLE PENTOPNON ADD (field91
NUMBER(10),field92 NUMBER(10),field93 NUMBER(10),field94
NUMBER(10),field95 NUMBER(10),field96 NUMBER(10),field97
NUMBER(10),field98 NUMBER(10),field99 NUMBER(10),field100
NUMBER(10))",218);
/* Prepare create table statement */
rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 218,
      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
/* Execute the SQL statment */
rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
      (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
      OCI_DEFAULT);
strncpy(sql_string, "ALTER TABLE PENTOPNON ADD (bfield2
NUMBER(10),bfield3 NUMBER(10),bfield4 NUMBER(10),bfield5
NUMBER(10),bfield6 NUMBER(10),bfield7 NUMBER(10),bfield8
NUMBER(10),bfield9 NUMBER(10),bfield10 NUMBER(10))",199);
/* Prepare create table statement */
rc = OCIStmtPrepare(p_sql, p_err, sql_string,(ub4) 199,
      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
/* Execute the SQL statment */
rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) 1, (ub4) 0,
      (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
      OCI_DEFAULT);
/* Prepare insert statement */
rc = OCIStmtPrepare(p_sql, p_err,"insert into pentopnon
      (pk1,pk2,field3,field4,..,field99,field100,bfield2,..,
      bfield10) values (:c1..,:c109)",(ub4) 1437,
      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
printf("Start of loop\n");
j=1;
c_init=clock()/1000.0;  // Take initial time reading

for(i2=1;i2<=N;i2++){  //Generate N records
      i=((i2-1)%Narray)+1;
      for (k=0;k<=108;k++){  // Generate 109 fields of data
            if(k==0)
                  a[0][i-1]=i2;
            if(k==1)
                  a[1][i-1]=j;
            if(k>=2 && k<=99)
                  a[k][i-1]=100000+100*(k+1)+i2;
            if(k>99)
                  a[k][i-1]=200000+100*(k-98)+j;
            if(i==Narray&&i2==Narray){
/* Do a bulk bind of the data here. Since an array is being
bound, OCIBindArrayOfStruct must be used */
            rc = OCIBindByPos(p_sql, &p_bnd, p_err,k+1,
            (dvoid *) &a[k][0], sizeof(int), SQLT_INT,
            (dvoid *) 0,(ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
            OCI_DEFAULT);
            rc = OCIBindArrayOfStruct(p_bnd,p_err,(ub4) 4,
            (ub4) 0,(ub4) 0,(ub4) 0);
            }
      }
      j++;
      if(j>N2){
            j=1;
```

```
        }

    if(i==Narray)  //Perform insertion for every Narray records
        rc = OCIStmtExecute(p_svc, p_sql, p_err, (ub4) Narray,
        (ub4) 0,(CONST OCISnapshot *) NULL, (OCISnapshot *) NULL,
        OCI_DEFAULT);
        if(i2%Ncommit==0)  // COMMIT every Ncommit records.
             OCITransCommit(p_svc, p_err, (ub4) 0);
        }

        /* Do a final commit in case N is not wholly divisible
        by Ncommit. */
        OCITransCommit(p_svc, p_err, (ub4) 0);

        c_time=clock()/1000.0;  // Take final timestamp
        printf("%10.3f\n",c_time-c_init);  // Print elapsed time

        /* Disconnect and free handles */
        rc = OCILogoff(p_svc, p_err);
        rc = OCIHandleFree((dvoid *) p_sql, OCI_HTYPE_STMT);
        rc = OCIHandleFree((dvoid *) p_svc, OCI_HTYPE_SVCCTX);
        rc = OCIHandleFree((dvoid *) p_err, OCI_HTYPE_ERROR);

        fclose(fp);

        return;
    }
```

DISTRIBUTION LIST

Evaluation of Methods for Rapidly Inserting Data into an Oracle Relational Database

Mark I. Porter and Ian L. Coat

AUSTRALIA

DEFENCE ORGANISATION

Task Sponsor
Defence Signals Directorate
Felicity Lawrence

S&T Program
Chief Defence Scientist
FAS Science Policy
AS Science Corporate Management
Director General Science Policy Development
} shared copy
Counsellor Defence Science, London (Doc Data Sheet only)
Counsellor Defence Science, Washington (Doc Data Sheet only)
Scientific Adviser Joint
Navy Scientific Adviser (Doc Data Sheet and distribution list only)
Scientific Adviser - Army (Doc Data Sheet and distribution list only)
Air Force Scientific Adviser (Doc Data Sheet and distribution list only)

Scientific Adviser to the DMO (Doc Data Sheet and distribution list only
Director of Trials

Information Sciences Laboratory

Chief of Intelligence, Surveillance and Reconnaissance Division (Doc Data Sheet and Distribution List Only)
Research Leader Secure Communications Branch (Doc Data Sheet and Distribution Sheet Only)
Head Communications Analysis
Mark I. Porter
Ian L. Coat

DSTO Library and Archives
Library Edinburgh 1 copy and Doc Data Sheet
Australian Archives

Capability Systems Division
Director General Maritime Development (Doc Data Sheet only)
Director General Aerospace Development (Doc Data Sheet only) Director General Information Capability Development (Doc Data Sheet only)

Office of the Chief Information Officer
Chief Information Officer (Doc Data Sheet only) (
Deputy CIO (Doc Data Sheet only)

Director General Information Policy and Plans (Doc Data Sheet only)
AS Information Structures and Futures (Doc Data Sheet only)
AS Information Architecture and Management (Doc Data Sheet only)
Director General Australian Defence Information Office (Doc Data Sheet only)
Director General Australian Defence Simulation Office (Doc Data Sheet only)

Strategy Group
Director General Military Strategy (Doc Data Sheet only)
Director General Preparedness (Doc Data Sheet only))

HQAST
SO (ASJIC) (Doc Data Sheet only))

Navy
SO (SCIENCE), COMAUSNAVSURFGRP, NSW (Doc Data Sheet and distribution list only)

Army
ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal (4 copies)
SO (Science), Deployable Joint Force Headquarters (DJFHQ) (L), Enoggera QLD (Doc Data Sheet only)
SO (Science) - Land Headquarters (LHQ), Victoria Barracks NSW (Doc Data Sheet and Executive Summary Only)

Air Force
Director General Policy and Plans, Air Force Headquarters (Doc Data Sheet only)
Director General Technical Air Worthiness, RAAF Williams (Doc Data Sheet only)
Chief of Staff – Headquarters Air Command, RAAF Glenbrook (Doc Data Sheet only)
Commander Aircraft Research and Development Unit, RAAF Edinburgh (Doc Data Sheet only)
Commander Air Combat Group, RAAF Williamtown (Doc Data Sheet only)
Staff Officer (Science), RAAF Amberley (Doc Data Sheet only)
Staff Officer (Science), RAAF Williamtown (Doc Data Sheet only)
Commander Air Lift Group, RAAF Richmond (Doc Data Sheet only)
Comander Maritime Patrol Group, RAAF Edinburgh (Doc Data Sheet only)
Commander Surveillance Control Group, RAAF Williamtown (Doc Data Sheet only)
Commander Combat Support Group, RAAF Amberley (Doc Data Sheet only)
Commander Training, RAAF Williams (Doc Data Sheet only)

Intelligence Program
DGSTA Defence Intelligence Organisation
Manager, Information Centre, Defence Intelligence Organisation
Assistant Secretary Corporate, Defence Imagery and Geospatial Organisation (Doc Data Sheet only)

Defence Materiel Organisation
Head Airborne Surveillance and Control (Doc Data Sheet only)
Head Aerospace Systems Division (Doc Data Sheet only)
Head Electronic Systems Division (Doc Data Sheet only)
Head Maritime Systems Division (Doc Data Sheet only)

Head Land Systems Division (Doc Data Sheet only)

Defence Libraries
Library Manager, DLS-Canberra (Doc Data Sheet Only)
Library Manager, DLS - Sydney West (Doc Data Sheet Only)

UNIVERSITIES AND COLLEGES
Australian Defence Force Academy
Library
Head of Aerospace and Mechanical Engineering
Hargrave Library, Monash University (Doc Data Sheet only)
Librarian, Flinders University

OTHER ORGANISATIONS
National Library of Australia
NASA (Canberra)
State Library of South Australia

OUTSIDE AUSTRALIA

INTERNATIONAL DEFENCE INFORMATION CENTRES
US Defense Technical Information Center, 2 copies
UK Defence Research Information Centre, 2 copies
Canada Defence Scientific Information Service, 1 copy
NZ Defence Information Centre, 1 copy

ABSTRACTING AND INFORMATION ORGANISATIONS
Library, Chemical Abstracts Reference Service
Engineering Societies Library, US
Materials Information, Cambridge Scientific Abstracts, US
Documents Librarian, The Center for Research Libraries, US

INFORMATION EXCHANGE AGREEMENT PARTNERS
Acquisitions Unit, Science Reference and Information Service, UK

SPARES (5 copies)

Total number of copies: **37**

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Evaluation of Methods for Rapidly Inserting Data into an Oracle Relational Database

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Mark I. Porter and Ian L. Coat

5. CORPORATE AUTHOR

Information Sciences Laboratory
PO Box 1500
Edinburgh South Australia 5111 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TN-0487	AR- 012-722	Technical Note	April 2003

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
E8730/16/60	INT 01/325	POLCOM	18	9

13. URL on the World Wide Web

http://www.dsto.defence.gov.au/corporate/reports/DSTO-TN-0487.pdf

14. RELEASE AUTHORITY

Chief, Intelligence, Surveillance and Reconnaissance Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS No

18. DEFTEST DESCRIPTORS

Relational databases, Data storage systems, ORACLE (computer system), Database management

19. ABSTRACT

In the Intelligence, Surveillance and Reconnaissance Division (ISRD) there are numerous applications in which large volumes of data are collected and analysed. Often, relational databases are used to store such data, as they improve certain analytical tasks through querying. In the literature, there is ample documentation of methods by which queries on a database may be optimised. However, there is scant information on the optimal technique for the initial insertion of data at high speed. This paper addresses the problem by investigating numerous insertion methods and comparing their performance for a given data set. The work was carried out on standard PCs running a commercially available database product (Oracle) and using common languages (such as Java, C and PL/SQL), and therefore may be of interest to the wider DSTO community.